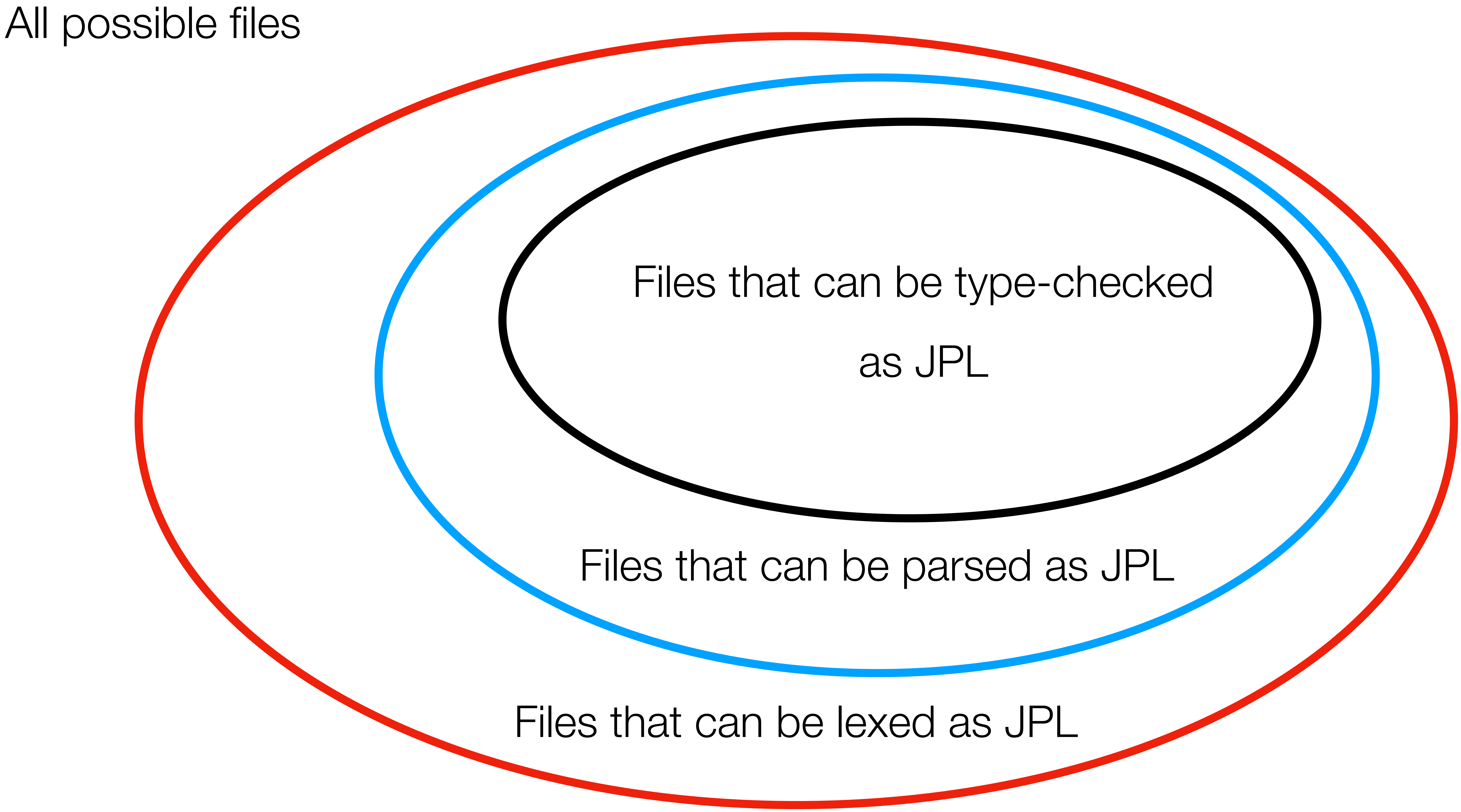
Intro to Parsing

Review

- The lexer takes an arbitrary file and turns it into:
    1. A lexer error
    2. A list of tokens
- The idea is to impose a first level of structure onto a file
- Divide-and-conquer approach to compiler construction!

Parsing

- The parser takes a list of tokens and turns it into one of:
    1. A parser error
    2. An abstract syntax tree (AST)
- The idea is to impose a second level of structure onto a file
- Once you have the AST, you never look at the list of tokens again



All possible files

Files that can be type-checked as JPL

Files that can be parsed as JPL

Files that can be lexed as JPL

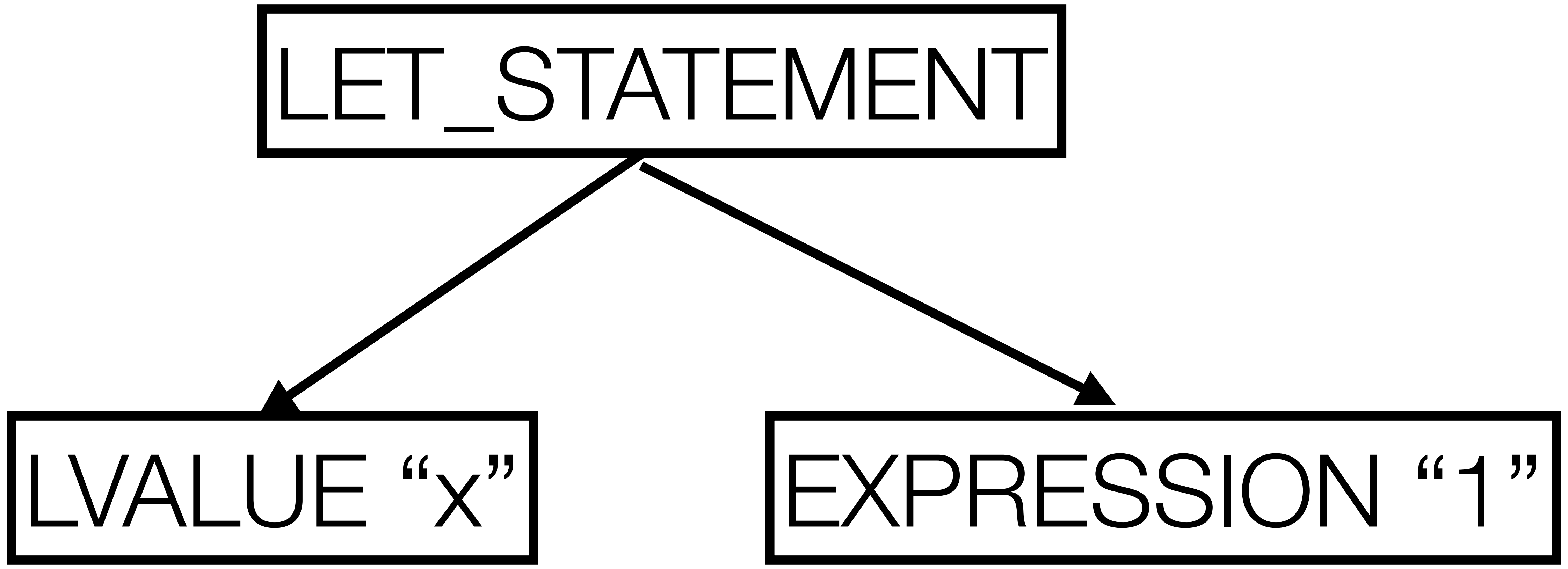
Let’s Parse

- File contains: **`let x = 1`**
- Lexer produces
  1. LET
  2. VARIABLE “x”
  3. EQUALS
  4. INTVAL “1”

Let’s Parse

- File contains: `let x = 1`
- Lexer produces
    1. LET
    2. VARIABLE “x”
    3. EQUALS
    4. INTVAL “1”

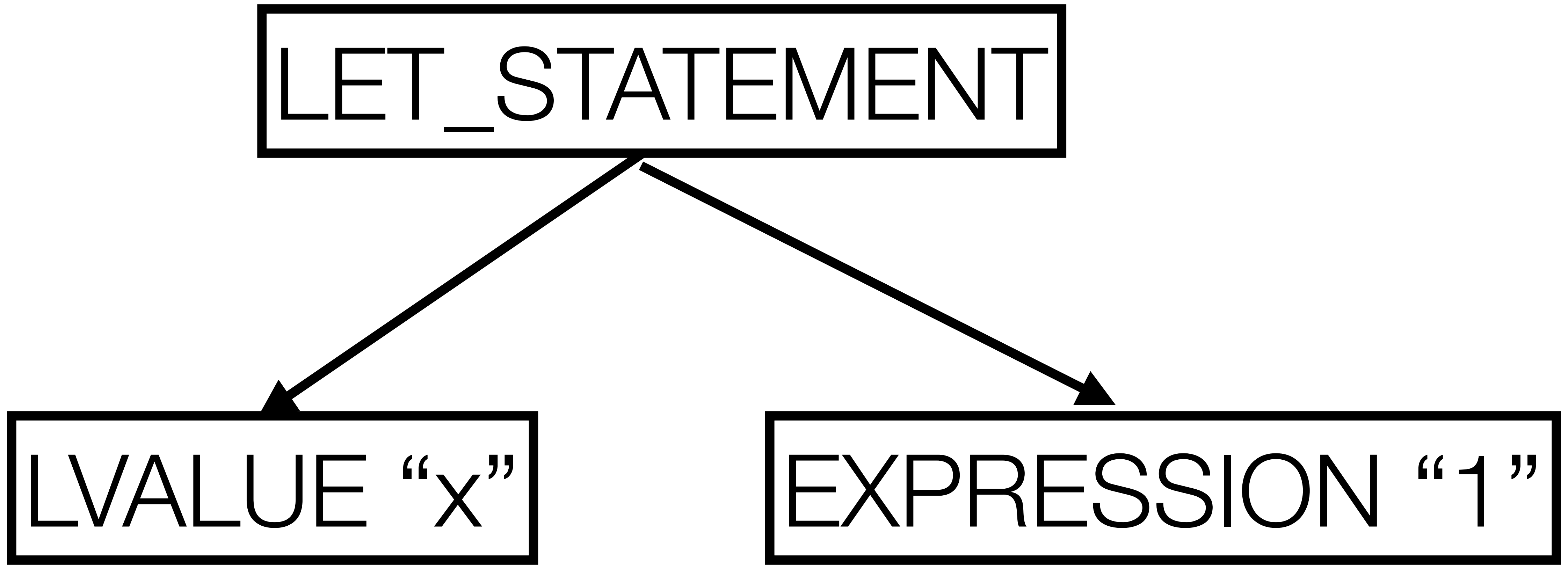
- This is the AST we want:



Let’s Parse

Where did the “=“ go??

- File contains: `let x = 1`
- Lexer produces
  1. LET
  2. VARIABLE “x”
  3. EQUALS
  4. INTVAL “1”
- This is the AST we want:



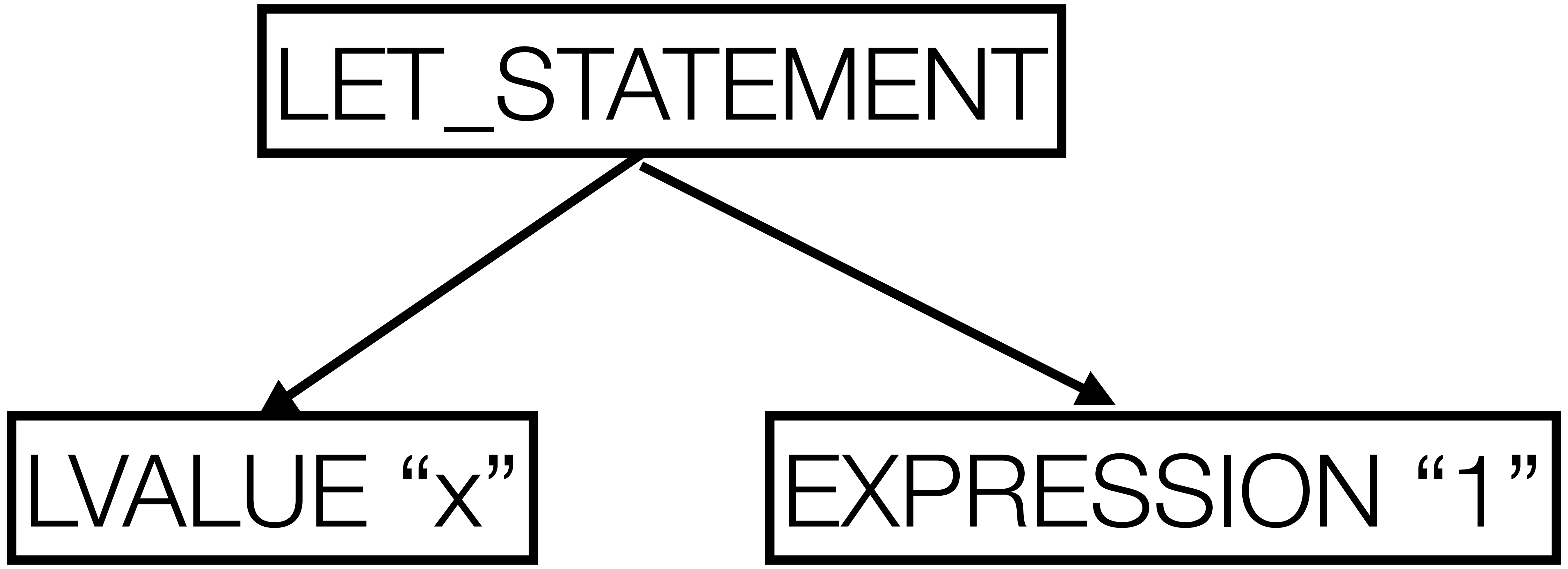
Let’s Parse

Where did the “=“ go??

- File contains: `let x = 1`
- Lexer produces
  1. LET
  2. VARIABLE “x”
  3. EQUALS
  4. INTVAL “1”

- This is the AST

How did we know to make the AST in this shape?



LET_STATEMENT

LVALUE “x”

EXPRESSION “1”

Let’s Parse

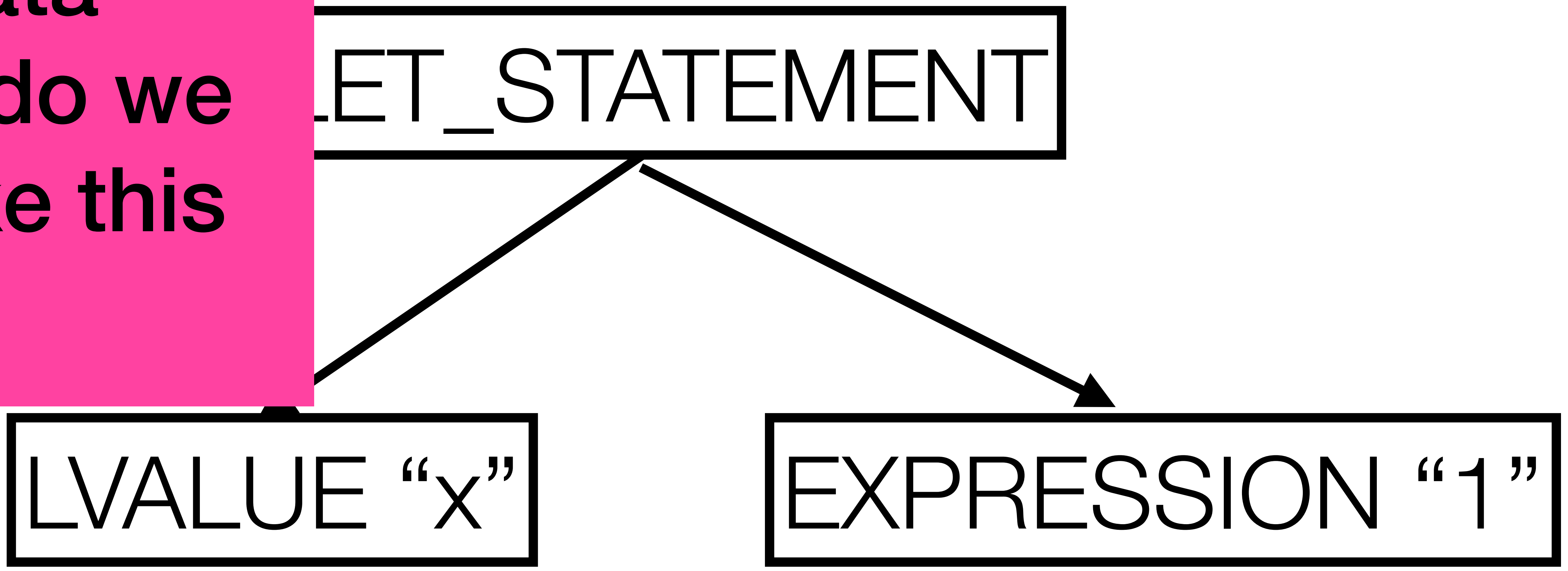
Where did the “=“ go??

- File contains: **`let x = 1`**
- Lexer produces
  1. LET
  2. VARIABLE “x”
  3. EQUALS
  4. INTVAL “1”

- This is the AST

How did we know to make the AST in this shape?

What data structures do we use to make this AST?



Let's Parse

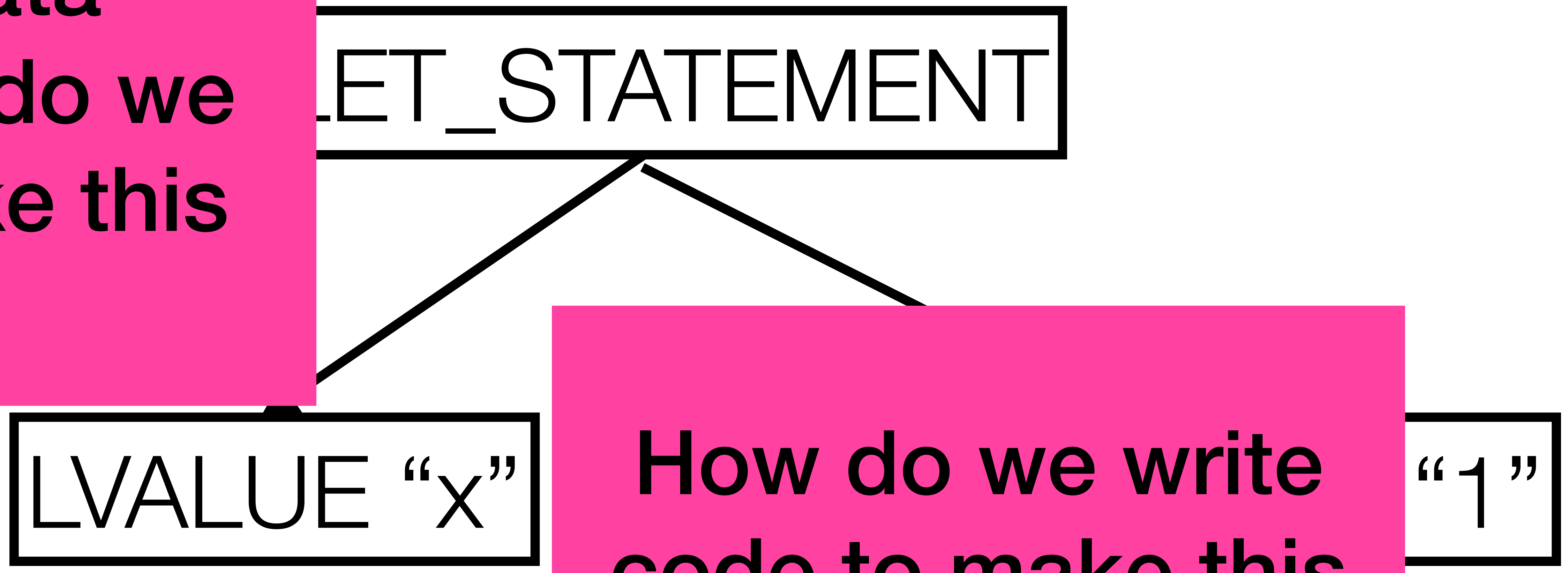
Where did the "=" go??

- File contains: `let x = 1`
- Lexer produces
  1. LET
  2. VARIABLE "x"
  3. EQUALS
  4. INTVAL "1"

- This is the AST

How did we know to make the AST in this shape?

What data structures do we use to make this AST?



LET_STATEMENT

LVALUE "x"

"1"

How do we write code to make this AST?

Let’s Parse

- File contains: **`let x = 2 * y`**
- Lexer produces:
    - LET, VARIABLE “x”, EQUALS, INTVAL “2”, OP “*”, VARIABLE “y”

- Parser pseudocode:

1. Someone calls a function recognize_let_statement()
2. recognize_let_statement() calls

    A. expect_token(LET)

    B. recognize_lvalue()

    C. expect_token(EQUALS)

    D. recognize_expression()

Let’s Pars

- File contains: `le`
- Lexer produces:
    - LET, VARIABL
      INTVAL “2”, OP “*”, VARIABLE “y”

Each of these steps consumes one or more tokens from the token list

- Parser pseudocode:
    1. Someone calls a function recognize_let_statement()
    2. recognize_let_statement() calls
        A. expect_token(LET)
        B. recognize_lvalue()
        C. expect_token(EQUALS)
        D. recognize_expression()

Let’s Parse

- File contains: **`let x = 2 * y`**
- Lexer produces:
    - LET, VARIABLE “x”, EQUALS, INTVAL “2”, OP “*”, VARIABLE “y”

- Parser pseudocode:

```
1. recognize_expression() calls
    A. recognize_int_value()
    B. expect_token(OP)
    C. recognize_variable()
2. Then it returns back to its caller
```

- What if, instead of “y”, the input contained “7” at that position?
- What if, instead of “y”, the input contained “foo(y)” at that position?

Let’s Parse

- File contains: **`let x = let * y`**
- Lexer produces:
  - LET, VARIABLE “x”, EQUALS, LET, OP “*”, VARIABLE “y”
- What happens when we try to parse this token list?

Recursive Descent Parsing

- The language grammar is recursively defined
- The AST is a recursive data structure mirroring the structure of the grammar
- The parser is a recursive algorithm whose structure mirrors both the AST and the grammar

Recursive Descent Parsing

- Many real compilers use hand-written recursive-descent parsers
    - For example, GCC and Clang
- Other parsing algorithms exist!
    - They are very hard to write by hand
    - Mostly, these algorithms are used by parser generators
    - We are not using parser generators in this class

Recursive Descent

- Keep in mind the first rule of recursion: Every recursive loop must have a **variant**
  - A loop invariant is a property that provably remains unchanged across iterations: We use these to prove loops correct
  - A loop variant is a property that provably changes across iterations: We use these to prove that loops terminate
- The usual variant in recursive descent parsers is:
  - “Every recursive loop must consume at least one token”
  - If this is not the case, your parser is likely to get stuck in an infinite loop